In-band Network Telemetry (INT)

October 17, 2017

Changhoon Kim, Jeongkeun Lee, Masoud Moshref, Mickey Spiegel, Ed Doe: *Barefoot Networks*
Dennis Cai: *Alibaba*
Hugh Holbrook: *Arista*
Anoop Ghanwani: *Dell*
Dan Daly: *Intel*
Mukesh Hira, Bruce Davie: *VMware*

1. Introduction

Inband Network Telemetry ("INT") is a framework designed to allow the collection and reporting of network state, by the data plane, without requiring intervention or work by the control plane. In the INT architectural model, packets contain header fields that are interpreted as "telemetry instructions" by network devices. These instructions tell an INT-capable device what state to collect and write into the packet as it transits the network. INT ***traffic sources*** (applications, end-host networking stacks, hypervisors, NICs, send-side ToRs, etc.) can embed the instructions either in normal data packets or in special probe packets. Similarly, INT ***traffic sinks*** retrieve (and optionally report) the collected results of these instructions, allowing the traffic sinks to *monitor the exact data plane state that the packets "observed" while being forwarded.*
Some examples of traffic sink behavior are described below:

- OAM – the traffic sink might simply collect the encoded network state, then export that state to an external controller. This export could be in a raw format, or could be combined with basic processing (such as compression, deduplication, truncation).
- Real-time control or feedback loops – traffic sinks might use the encoded data plane information to feed back control information to traffic sources, which could in turn use this information to make changes to traffic engineering or packet forwarding. (Explicit congestion notification schemes are an example of these types of feedback loops).
- Network Event Detection - If the collected path state indicates a condition that requires immediate attention or resolution (such as severe congestion or violation of certain data-plane invariances), the traffic sinks could generate immediate actions to respond to the network events, forming a feedback control loop either in a centralized or a fully decentralized fashion (a la TCP).

The INT architectural model is intentionally generic, and hence can enable a number of interesting high level applications, such as:

- Network troubleshooting
  - L1 traceroute, micro-burst detection, packet history (a.k.a. postcards), trajectory sampling
- Advanced congestion control
  - RCP, XCP, TIMELY
- Advanced routing
  - Utilization-aware routing (e.g., CONGA)
- Network data-plane verification

A number of use case descriptions and evaluations are described in the Millions of Little Minions paper.

2. Terms

INT Header: Any packet header that carries INT information. There are two types of INT Headers -- *Hop-by-hop* and *Destination* (See Section 4.1).

INT Packet: Any packet containing an INT Header.

INT Instruction: Embedded packet instructions indicating which INT Metadata to collect (defined below). The collected data is written into an INT Header.

INT Source: A trusted entity that creates and inserts INT Headers into the packets it sends. The INT Headers contain, at minimum, INT Instructions indicating what to collect.

INT Sink: A trusted entity that extracts the INT Headers and collects the path state contained in the INT Headers. The INT Sink is responsible for removing INT Headers so as to make INT transparent to upper layers. (Note that this does not preclude having nested or hierarchical INT domains.)

INT Transit Hop: A networking device that adds its own INT Metadata to an INT Packet by following the INT Instructions in the INT Header.

INT Metadata: Information that an INT Source or an INT Transit Hop device inserts into the INT Header. Examples of metadata are described below.

3. What To Monitor

In theory, one may be able to define and collect ***any*** switch-internal information using the INT approach. In practice, however, it seems useful to define a small baseline set of metadata that can be made available on a wide variety of devices: the metadata listed in this section comprises such a set. As the INT specification evolves and the INT technology becomes more popular, we expect to add more metadata to this INT specification.

Note the exact meaning of the following metadata (e.g., the unit of timestamp values, the precise definition of hop latency or congestion status) can vary by device for any number of reasons, including the heterogeneity of device architecture, feature sets, resource limits, etc. Thus, defining the exact meaning of each metadata is beyond the scope of this document. Instead we assume the users of INT will communicate the precise meanings of these metadata for each device model they use in their networks.

Each metadata is encoded as a 4B unsigned value.

3.1. Switch-level Information

- Switch id
    - The unique ID of a switch (generally administratively assigned). Switch IDs must be unique within a management domain.

3.2. Ingress Information

- Ingress port id
    - The logical port on which the INT packet was received. The semantics (meaning) of this value may differ for individual devices. A device may expose physical and logical port identifiers separately.
- Ingress timestamp
    - The device local time when the INT packet was received on the ***ingress*** physical or logical port.

3.3. Egress Information

- Egress port ID
    - The ID of the output port via which the INT packet was sent out. The exact meaning of this value can differ for individual devices, and defining that is beyond the scope of this document. For example, some devices may encode the physical port ID, whereas the other may encode the logical port ID. Some other types of devices can encode both (2B each).
- Hop latency
    - Time taken for the INT packet to be switched within the device.
- Egress port TX Link utilization
    - Current utilization of the egress port via which the INT packet was sent out. Again, devices can use different mechanisms to keep track of the current rate, such as bin bucketing or moving average. While the latter is clearly superior to the former, the INT framework does not stipulate the mechanics and simply leaves those decisions to device vendors.

3.4. Buffer Information

- Queue occupancy
    - The build-up of traffic in the queue (in bytes, cells, or packets) that the INT packet observes in the device while being forwarded.
- Queue congestion status
    - The fraction (in percentage or decimal fraction) of current queue occupancy relative to the queue-size limit. This indicates how much buffer space was used relative to the maximum buffer space (instantaneously or statically) available to the queue.

4. Processing INT Headers

4.1. INT Header Types

There are two types of INT Headers, ***hop-by-hop*** and ***destination***. A given INT packet may have either or both types of INT Headers. When both INT Header types are present, the hop-by-hop type must precede the destination type header.

- Hop-by-Hop type
    - Intermediate devices (INT Transit Hops) must process this type of INT Header.
- Destination type
    - Destination headers must only be consumed by the INT Sink; intermediate devices must ignore Destination headers.
    - Destination headers can be used for two purposes (for example):
        - To enable communication between INT Source and INT Sink.
            - INT Source can add a sequence number to detect lost INT packets.
            - INT Source can add the original IP TTL value of an INT packet. This way, an INT Sink can detect network devices on the path that do not support INT (and hence failed to add INT metadata) by checking the difference between the number of INT metadata instances in the INT Header (i.e., # of INT-compliant hops) and the decrement of the IP TTL values (i.e., # of physical IP hops).
        - To deliver follow-up INT packets to the INT Sinks (see Section 4.2)
            - Follow-up packets generated by the slow-path forwarding logic of a transit-hop switch must carry the original INT instructions but must not trigger any further INT processing by downstream devices.

4.2. Handling INT Packets

It is obviously preferable for network devices to process any INT packets strictly within the fast path, often a hardware based forwarding plane. An ideal system would be able to process INT instructions with no increase in latency or reduction in forwarding performance, but in some cases it may be required to process INT packets outside the fast path. This slow-path processing could be a CPU based control plane, some sideband or alternate hardware assisted forwarding path, or an arbitrary INT resource. Note that in the case where the INT processing is done outside the fast path, the device MUST still forward the original packet through the fast path (i.e. without processing the INT instructions). The implementation of this is not specified, though it implies the ability to make a copy of the INT packet for slow-path processing or a similar design. Following the processing of the INT packet in the normal fast-path, the forwarding plane should generate a ***trigger*** toward the slow path (e.g., either a copy of the original INT packet or a digest of it). Upon receiving the trigger, the slow path should process the INT Header appropriately, generate a new packet -- called a “follow-up packet” -- containing the execution results of the INT instructions. The follow-up packet is forwarded separately.

If devices in the network do perform slow-path INT processing, it is possible that a single INT packet could spawn multiple follow-up packets – and in turn each of these could spawn more INT processing downstream. Care must be taken to prevent excessive replication. To prevent the cascading generation of follow-up telemetry packets, all follow-up packets are marked with a special "exemption" flag. The presence of this flag instructs downstream devices to provide specific processing. For more specific information, see the option type for INT Source-to-sink communication messages (Destination type in Section 4.1).

The INT Header of a follow-up packet must contain all the existing INT metadata in the original packet that was added by the upstream devices, as well as its own local INT metadata. Follow-up packets must contain enough information (from outer header, inner header, or both of the original packet) so that the INT sink can correlate the follow-up packets with the original INT packet. Because INT is not tied to a particular encapsulation protocol, this spec does not dictate the exact format of a follow-up packet other than its INT portion.

To prevent potential packet ordering issues, it is recommended that an INT device ***NOT*** forward the INT packets themselves via the slow path while processing INT Headers.

5. Header Format and Location

This section specifies the format and location of INT Headers.

5.1. INT over any encapsulation

The specific location (i.e. encapsulation header) for INT Headers are intentionally NOT specified -- an INT Header can be inserted as an option or payload of ***any*** encapsulation type. The only requirements are that encapsulation header provides sufficient space to carry the INT information and that both the INT Sources and Sinks can agree on the location of the INT Headers. The following choices are all potential encapsulations using common protocol stacks, although the INT user may choose a different encapsulation format if better suited to their needs and environment.

- INT over VXLAN (as a VXLAN payload, per GPE extension)
- INT over Geneve (as a Geneve option)
- INT over NSH (as a NSH payload)
- INT over TCP (as payload)
- INT over UDP (as payload)

For each encapsulation format, we need to reserve a next-header type identifier (e.g., a VXLAN-GPE Next Protocol value, a Geneve Option Class value, or a TCP/UDP port number) to indicate the presence of an INT Header.

As illustrative examples, we describe three encapsulation formats, both suitable for use in virtualized data centers:

1. INT over TCP/UDP - This example introduces a shim header after TCP/UDP header and carry INT Headers between the shim header and TCP/UDP payload. This approach doesn't rely on any tunneling/virtualization mechanism and is versatile to apply INT to both native and virtualized traffic.
2. *INT over VXLAN* - Our examples use the VXLAN generic protocol extensions (draft-ietf-nvo3-vxlan-gpe) to carry INT Headers between VXLAN header and encapsulated VXLAN payload.
3. *INT over Geneve* - Geneve is an extensible tunneling framework, allowing Geneve options to be defined for INT Headers.

5.2. On-the-fly Header Creation

In the INT model, each device in the packet forwarding path creates additional space in the INT Header on-demand to add its own INT metadata. To avoid exhausting header space in the case of a forwarding loop or any other anomalies, it is strongly recommended to limit the number of total INT metadata fields added by devices.

As with any modification that potentially changes the packet size, this on-the-fly allocation may "grow" a packet beyond the original MTU, resulting in fragmentation. It is advisable to plan network/NIC/MTU settings accordingly to leave headroom for the additional headers.

5.3. Header Format

This subsection proposes the INT Header format. Where necessary, we use examples based on the INT-over-TCP/UDP, INT-over-VXLAN or INT-over-GENEVE encapsulation formats.

5.3.1. Header Location and Format -- INT over TCP/UDP

In case the packet is not encapsulated by any virtualization header, INT over VXLAN or INT over GENEVE is not helpful. Instead, one can put the INT metadata just after layer 4 headers (TCP/UDP). If TCP has any options, the INT stack comes after the TCP options. The scheme assumes that the non-INT devices between the INT source and the INT sink either do not parse beyond layer-4 headers or can skip through the INT stack using the Length field of INT shim header.
(Note: INT over UDP can be used even when the packet is encapsulated by VXLAN, GENEVE, or GUE (Generic UDP Encapsulation). INT over TCP/UDP also makes it easier to add INT stack into outer, inner, or even both layers. In such cases both INT header stacks carry information for respective layers and need not be considered interfering with each other.)

A field in the lower layers of Ethernet, IP, or TCP/UDP should indicate if the INT header exists after the TCP/UDP header. We propose two options:

- IPv4 DSCP or IPv6 Traffic Class field: A value or a bit will be reserved to indicate the existence of INT after TCP/UDP. INT source will put the reserved value in the field, and INT sink will remove it. The source can store the original DSCP so that the sink can restore the original value. Restoring the original value is optional. (Note: allocating a bit, as opposed to a value codepoint, will allow the rest of DSCP field to be used for QoS, hence allowing the coexistence

of DSCP-based QoS and INT. The QoS engine must be programmed to ignore the designated bit position.)

- TCP/UDP destination Port field: A port number in layer 4 will be reserved to indicate the existence of INT after TCP/UDP. INT source changes the destination Port, and sink must restore the original port number.

The decision to use DSCP or destination port field and the reserved values are per domain decisions that all INT capable devices should follow. (Note: in general, encoding into DSCP field will be less intrusive compared to changing the layer 4 port field. The latter may alter ECMP behavior and can complicate ACL and network debugging.)

INT over TCP/UDP adds INT metadata after TCP/UDP headers as if the payload is changed. However, the sink device will remove INT headers before passing the packet to the traffic destination. Therefore, updating the TCP/UDP checksum (and UDP.length in case of UDP) is optional and a per domain decision.

We introduce INT shim header and INT tail header for TCP/UDP, and their formats are as follows. The INT metadata header and INT metadata stack will be encapsulated between the shim and tail headers.

```
INT shim header for TCP/UDP:
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      Type     |    Reserved   |     Length    |    Reserved   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Type: This field indicates the type of INT Header following the shim header. Two Type values are used: one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).
- Length: This is the total length of INT metadata header, INT stack and the shim and tail headers in 4-byte words. A device can read this field and just skip through the whole INT headers to reach TCP/UDP payload

```
INT tail header for TCP/UDP:
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      Proto    |      Destination Port         | Reserved/DSCP |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Proto: A copy of IPv4.proto or IPv6.nextHeader.
- Destination Port: The original destination port of TCP/UDP protocol. In case the special destination port is used to indicate INT existence, this field is used by INT sink to restore the original value. Besides, having the proto and destination port fields in the tail header would help transit and sink devices to know how to parse the rest of the packet after the INT headers.
- Reserved/DSCP: In case the IP DSCP is used to indicate INT existence, this field stores the original DSCP value.

Proto and Destination Port fields are needed in the tail header regardless of which of DSCP or Destination Port was used to indicate the existence of INT.

5.3.2. Header Location and Format -- INT over VXLAN GPE

VXLAN is a common tunneling protocol for network virtualization and is supported by most software virtual switches and hardware network elements. The VXLAN header as defined in RFC 7348 is a fixed 8-byte header as shown below.

```
VXLAN Header:

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |R|R|R|R|I|R|R|R|                   Reserved                    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                VXLAN Network Identifier (VNI) |   Reserved    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The amount of free space in the VXLAN header allows for carrying minimal network state information. Hence, we embed INT metadata in a shim header between the VXLAN header and the encapsulated payload. This is the recommended approach as it allows for carrying more network state information along an entire path.

The VXLAN header as defined in RFC 7348 ***does not*** specify the protocol being encapsulated and assumes that the payload following the VXLAN header is an Ethernet payload. Internet draft draft-ietf-nvo3-vxlan-gpe-00.txt proposes changes to the VXLAN header to allow for multi-protocol encapsulation. We use this VXLAN generic protocol extension draft and propose a new “Next-protocol” type for INT.

```
VXLAN GPE Header:

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |R|R|R|R|I|P|R|R|           Reserved            | Next Protocol |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                VXLAN Network Identifier (VNI) |   Reserved    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

P bit: Flag bit 5 is defined as the Next Protocol bit. The P bit MUST be set to 1 to indicate the presence of the 8-bit next protocol field.

Next Protocol Values:
0x01: IPv4
0x02: IPv6
0x03: Ethernet

0x04: Network Service Header (NSH)
0x05: In-band Network Telemetry (INT) Header (the value is subject to change)

When there is one INT Header in the VXLAN GPE stack, the VXLAN GPE header for the INT Header will have a next-protocol value other than INT Header indicating the payload following the INT Header - typically Ethernet. If there are multiple INT Headers in the VXLAN GPE stack, then all VXLAN GPE **shim** headers for the INT Headers other than the last one will carry 0x05 for their next-protocol values. And, the VXLAN GPE header for the last INT Header will carry a next-protocol value of the original VXLAN payload (e.g., Ethernet).

To embed a variable-length data (i.e., INT metadata) in the VXLAN GPE stack, we introduce the INT shim header of which format is as follows. This header follows each VXLAN GPE header for INT.

```
INT shim header for VXLAN GPE encapsulation:
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      Type     |    Reserved   |     Length    | Next-Protocol |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   Variable Option Data (INT Metadata Headers and Metadata)    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Type: This field indicates the type of INT Header following the shim header. Two Type values are used: one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).
- Length: This is the total length of the variable INT option data and the shim header in 4-byte words.

5.3.3. Header Location and Format -- INT over Geneve

Geneve is a generic and extensible tunneling framework, allowing for current and future network virtualization implementations to carry metadata encoded in TLV format as “Option headers” in the tunnel header.

```
Geneve Header:
   0                   1                   2                   3
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |Ver|  Opt Len  |O|C|    Rsvd.  |          Protocol Type        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |        Virtual Network Identifier (VNI)       |    Reserved   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Variable Length Options                    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Note we do not need to reserve any special values for fields in the base Geneve header for INT.

- Users may or may not use INT with Geneve along with VNI (network virtualization), though using INT with Geneve without network virtualization would be a bit wasteful.

```
Geneve Option Format:
   0                   1                   2                   3
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Option Class         |      Type     |R|R|R| Length  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   Variable Option Data (INT Metadata Headers and Metadata)    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- We need to reserve a unique Option Class value for INT.
- We need to reserve two Type values associated with the Option Class for INT -- one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).
- The variable length option data following the Geneve Option Header carries the actual INT metadata header and metadata.
- Note the Length field of the Geneve Option header is 5-bits long, which limits a single Geneve option instance to no more than 124 bytes long (31 * 4). If 124 bytes is insufficient one could collect different, non-overlapping sets of INT metadata info across multiple packets.

5.3.4. INT Metadata Header Format

In this section, we describe the format for INT metadata headers and the metadata itself.

```
INT Metadata Header and Metadata Stack:
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5   6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |Ver|Rep|C|E|R R R R R| Ins Cnt |  Max Hop Cnt  | Total Hop Cnt |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      Instruction Bitmap       |            Reserved           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    INT Metadata Stack (varying number of fixed-size values)   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                              . . .                            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                         Last INT metadata                     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Each INT metadata header is 8B long. While each metadata value is always encoded as a 4B-long value, the metadata stack following the INT metadata header has a varying length because different packets are delivered through different paths (and hence different number of hops).
- INT instructions are encoded as a bitmap in the 16-bit INT Instruction field: each bit corresponds to a specific standard metadata as specified in Section 3.
    - bit0 (MSB): Switch ID

    - bit1: Ingress port ID + egress port ID
    - bit2: Hop latency
    - bit3: Queue ID + Queue occupancy
    - bit4: Ingress timestamp
    - bit5: Egress timestamp
    - bit6: Queue ID + Queue congestion status
    - bit7: Egress port tx utilization
    - The remaining bits are reserved.
- Each INT Transit device along the path that supports INT adds its own metadata values as specified in the instruction bitmap immediately after the INT metadata header.
    - When adding a new metadata, each device must prepend its metadata in front of the metadata that are already added by the upstream devices. This is similar to the push operation on a stack. Hence, the most recently added metadata appears at the top of the stack.
    - If a device is unable to provide a metadata value specified in the instruction bitmap because its value is not available, it must add a special reserved value 0xFFFF_FFFF indicating “invalid”.
    - If a device cannot add all the number of bytes (i.e., Instruction Count * 4) required by the instruction bitmap (irrespective of the availability of the metadata values that are asked for), it must skip processing that particular INT packet entirely. This ensures that each INT Transit device adds ***either*** zero bytes or a fixed/well- known number of bytes to the INT packet.
- The fields in the INT metadata header are interpreted the following way:
    - Ver (2b): INT metadata header version. Should be zero for this version.
    - Rep (2b): Replication requested. Support for this request is optional. If this value is non-zero, the device may replicate the INT packet. This is useful to explore all the valid physical forwarding paths when multi-path forwarding techniques (e.g., ECMP, LAG) are used in the network. Note the Rep bits should be used judiciously (e.g., only for probe packets, not for every data packet). While we recommend that Rep bits be set only for probe packets, the INT architecture does not (and perhaps cannot) disallow use of the Rep bits for real data packets.
        - 0: No replication requested.
        - 1: Port-level (L2-level) replication requested. If the INT packet is forwarded through a logical port that is a port-channel (LAG), then replicate the packet on each physical port in the port-channel and send a single copy per physical port.
        - 2: Next-hop-level (L3-level) replication requested. Replicate the packet to each L3 ECMP next-hop valid for the destination address and send a single copy per ECMP next-hop.
        - 3: Port- and Next-hop-level replication requested.
    - C (1b): Copy.
        - If replication is indeed requested for data packets, the INT Sink must be able to distinguish the original packet from replicas so that it can forward only original packets up up the protocol stack, and drop all the replicas. The C bit must be set to 1 on each copy, whenever INT transit hop replicates a packet. The original packet must have C bit set to 0.
        - C bit must always be set to 0 by INT source
    - E (1b): Max Hop Count exceeded.

        - This flag must be set if a device cannot prepend its own metadata due to reaching the Max Hop Count. If the Total Hop Count of an incoming INT packet is identical to the Max Hop Count, the INT device cannot add its own INT metadata and must set this flag to 1.
    - R: Reserved bits.
    - Instruction Count (5b): The number of instructions that are set (i.e., number of 1's) in the instruction bitmap.
        - While the largest value of Instruction Count is 31, an INT-capable device may be limited in the maximum value of Instruction Count it supports, in which case it would cease processing an INT packet with a higher Instruction Count .
    - Max Hop Count (8b): The maximum number of hops that are allowed to add their metadata to the packet. If Total Hop Count (see below) equals Max Hop Count, a device must ignore the INT instruction, pushing no new metadata onto the stack.
    - Total Hop Count (8b): The total number of hops that added their metadata instance(s) to the INT packet.
        - The INT Source must set this value to zero upon creation of an INT metadata header, and each INT-capable device on the path must increment the Total Hop Count as it pushes its local metadata onto the stack.
- Summary of the field usage
    - The INT Source must set the following fields::
        - Ver, Rep, C, Instruction Count, Max Hop Count, Total Hop Count (0), and Instruction Bitmap
    - Intermediate devices can set the following fields:
        - C, E, Total Hop Count
- In an INT packet, the length (in bytes) of the INT metadata stack must always be (Instruction Count * 4 * Total Hop Count).

5.4. Examples

This section shows example INT Headers. The assumptions made for this example are as follows.

```
                ==> packet P travels from Host1 to Host2 ==>
   Host1 --------> Switch1 ---------> Switch2 ---------> Switch3 -------->* Host2
```

Two hosts (Host1 and Host2) communicate through a network path composed of three network devices -- Switch1, 2, and 3. The example in this section shows INT Headers attached to data packet P forwarded by Switch3 and received by Host2 (marked as `*` in the diagram above); the INT Source of this data packet is Host1, and the INT Sink is Host2. The INT metadata attached to this packet are the switch ID (sw id) and queue occupancy (q len) from every switch on the forwarding path from Host1 to Host2.

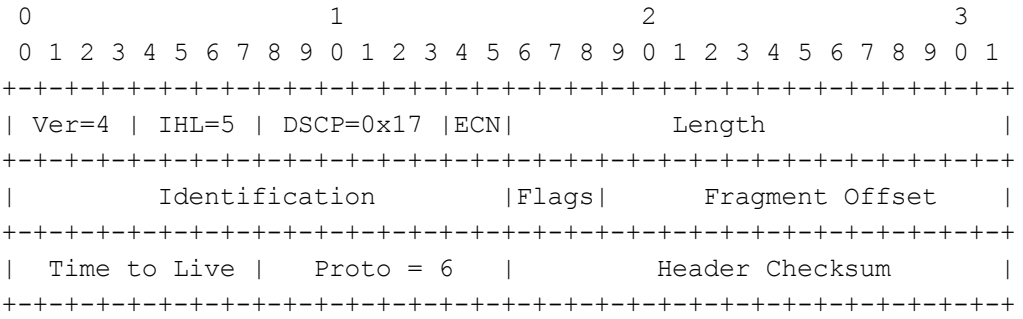
While it is not the scope of this spec, we assume in this particular example that Host1 also delivers the INT metadata (queue occupancy) collected via a previous packet it received from Host2 by piggybacking the data onto packet P. We assume that Host1 uses the destination-type INT Header for that because intermediate devices must ignore such a header type. We also assume that Host1 uses the same INT metadata header format for the piggybacked INT metadata just for convenience.

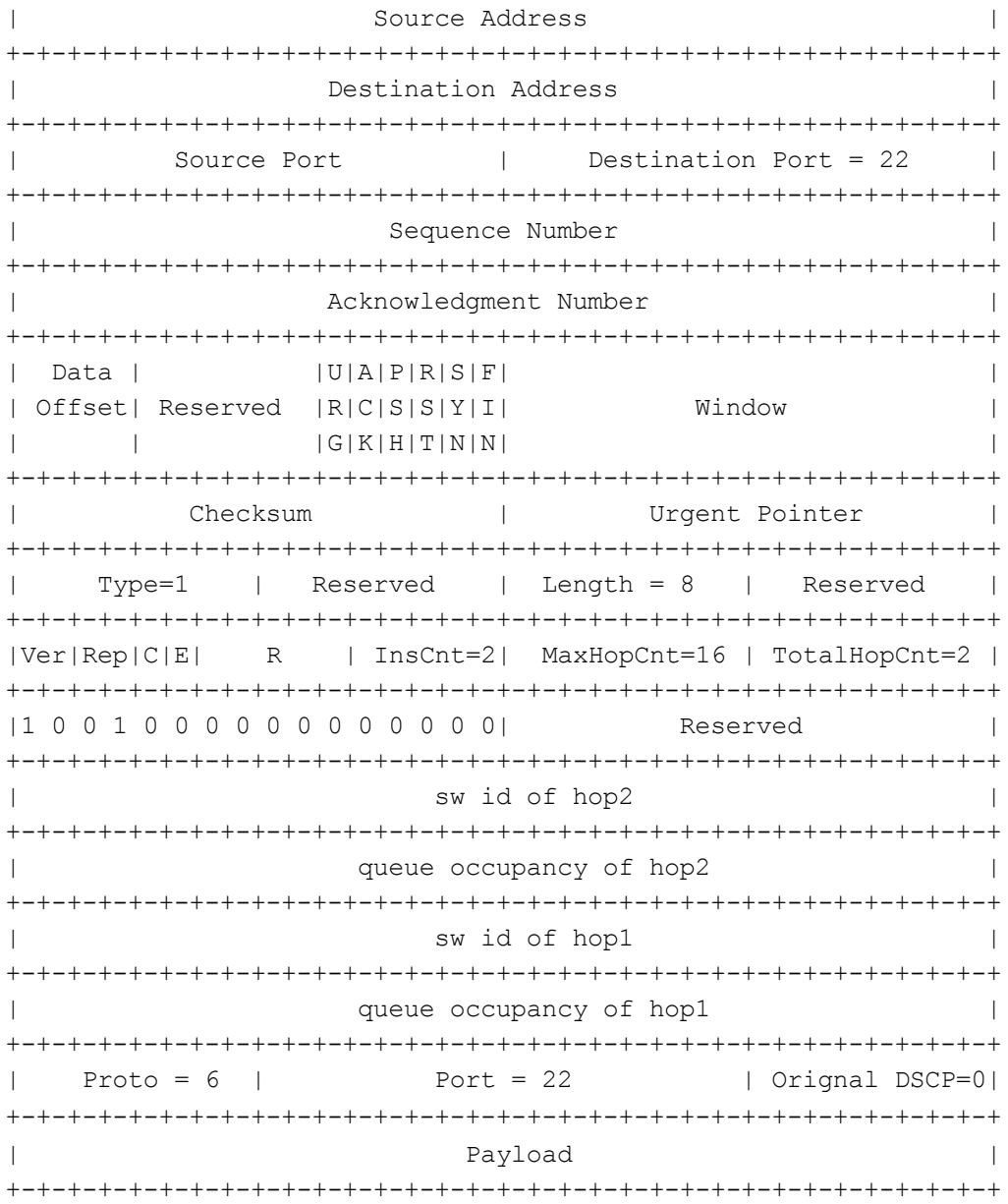
The following is the detailed assumption made for this example.

- As an INT Source, Host1 wants to collect switch id and queue occupancy from each device on the path. It uses the hop-by-hop-type INT Header to do so.
- As an INT Sink, Host1 wants to piggyback the queue occupancy values collected from the Host2-to-Host1 path onto the data packet sent back to Host2. It uses the destination-type INT Header.
- There are three devices (hops) on the path, and all the devices can expose both metadata (switch id and queue occupancy).
- The INT metadata header uses the following field values in the metadata header, unless specified otherwise in each example.
    - Ver = 0
    - Rep = 0 (No replication)
    - C = 0
    - E = 0 (Max Hop Count not exceeded)
    - Instruction Count = 2 (for switch id & queue occupancy)
    - Max Hop Count = 16 (network diameter)
- The piggybacked metadata header happens to use the same INT metadata header format with the following field values. Again, note this is only for example; we do not propose any designs for the piggybacked metadata format other than that this is out of the scope of this document and that we should reserve a special option type for this kind of data (i.e., INT Source-to-sink data).
    - Ver = 0
    - Rep = 0 (No replication)
    - C = 0
    - E = 0 (Max Hop Count not exceeded)
    - Instruction Count = 1 (for queue occupancy)
    - Max Hop Count = 16 (network diameter)

Example with INT over TCP

We now consider a scenario where host1 sends a TCP packet to host2. The ToR switch of host1 (hop1) adds INT over TCP. Then an aggregate switch adds hop2 information and finally ToR switch of host2 (hop3) receives the packet and plays the role of INT sink. Here is the packet received by INT sink starting from the IPv4 header. We use the value of 0x17 for IPv4.DSCP to indicate the existence of INT headers.

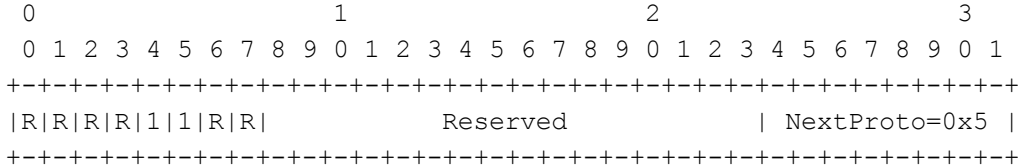


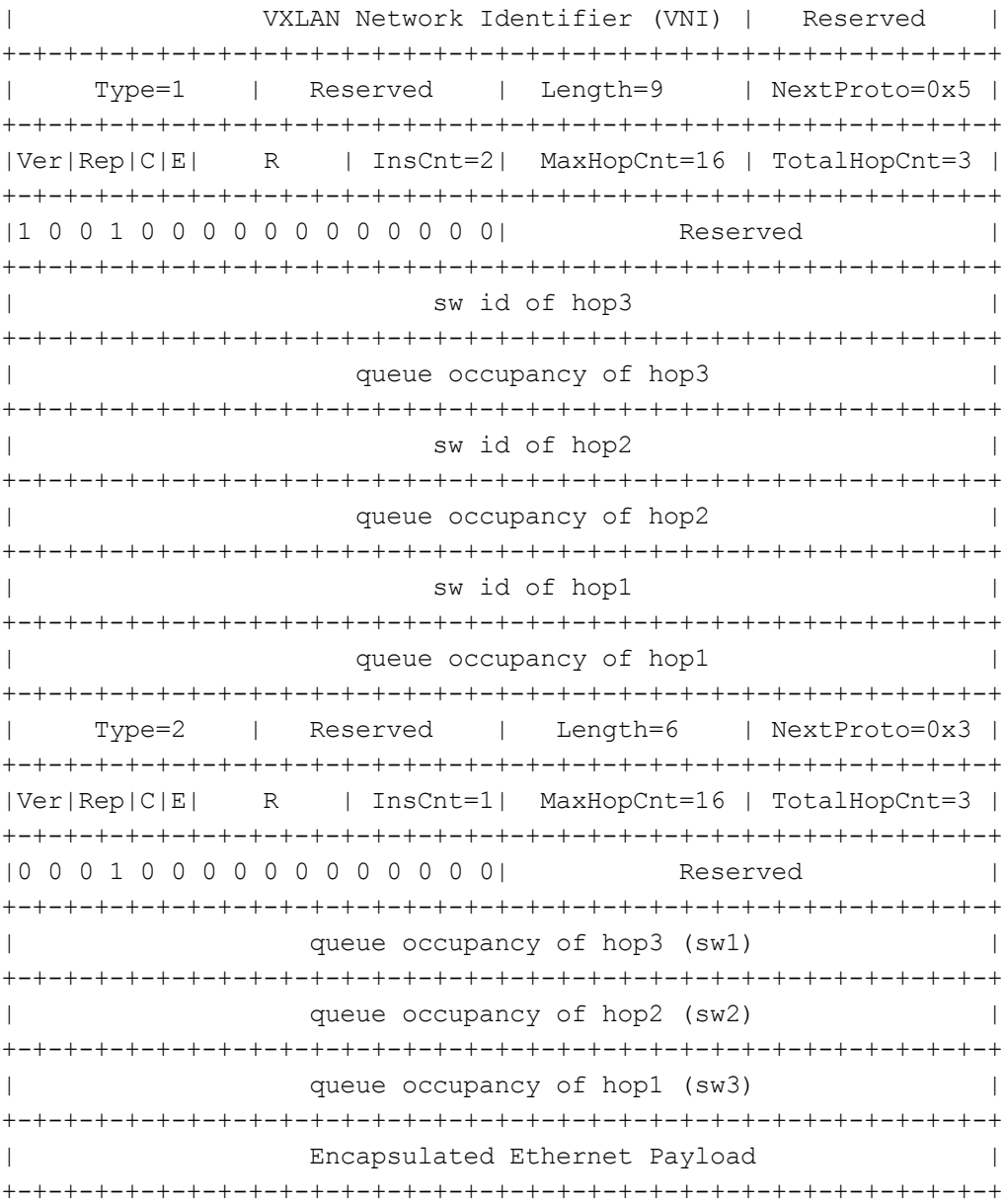


Example with VXLAN GPE encapsulation

We now consider a scenario where Host1 and Host2 use VXLAN encapsulation, intermediate switches parse through VXLAN header and the INT shim between VXLAN header and encapsulated payload and populate the INT metadata.

The packet headers received at Host 2 are as follows, starting with the VXLAN header (encapsulating ethernet, IP and UDP headers are not shown here):





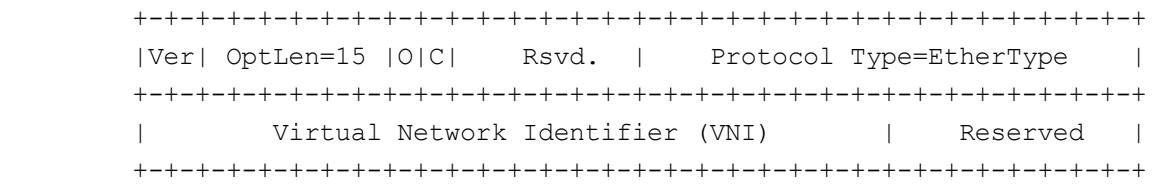
Example with Geneve encapsulation

We first consider a scenario where Host1 and Host2 are using Geneve encapsulation and the intermediate switches parse the Geneve headers and populate INT metadata.

We assume Geneve option class of 0x00AB for INT. We also assume that the Geneve option type value for the hop-by-hop INT Header type (i.e., the one that intermediate switches must process) is 1, and that the type value for the destination INT Header is 2.

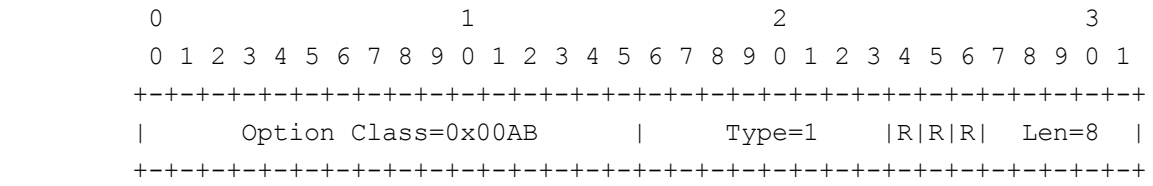
The following is the Geneve and INT Headers attached to the packet received by Host2.

```
Geneve Header:
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

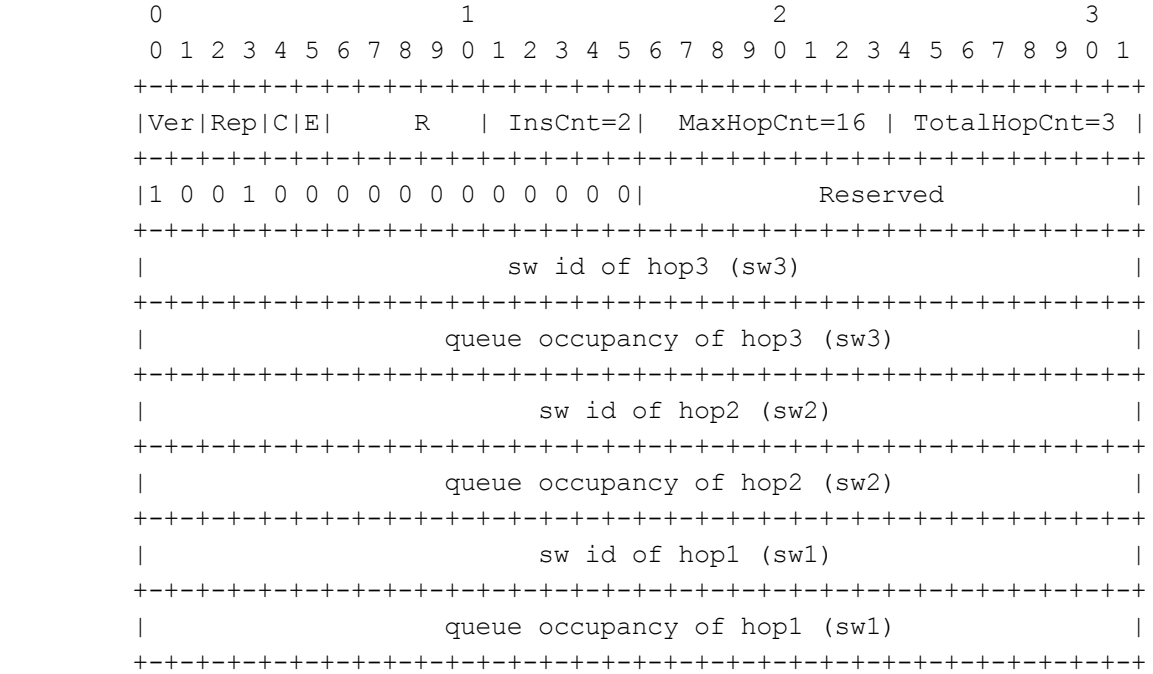


[INT Metadata - info about the path from Host1 to Host2]

Geneve Option for Switch ID metadata:

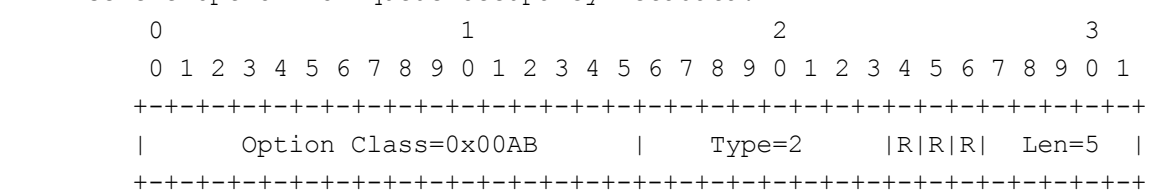


INT Metadata Header and Metadata Stack:

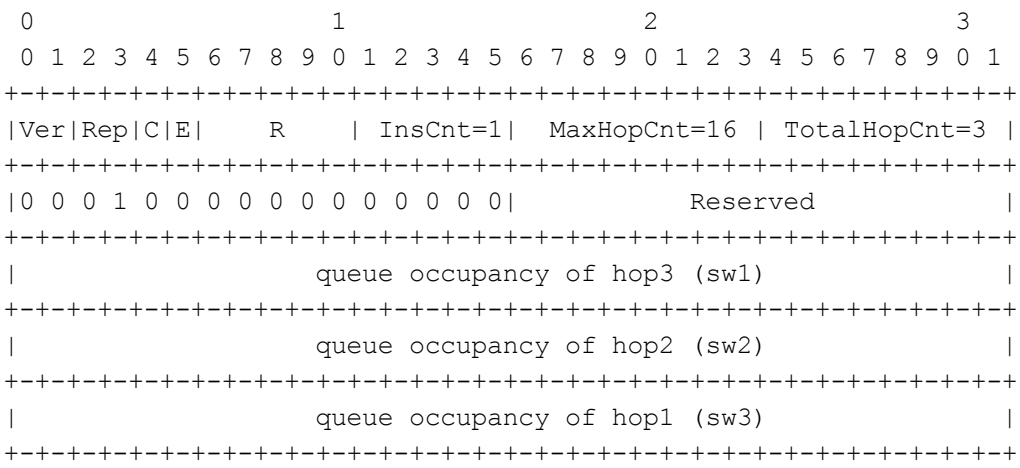


[Piggybacked metadata - info about the path from Host2 to Host1]

Geneve Option for queue occupancy metadata:



INT Metadata Header and Metadata Stack:


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Ver
Rep
C
E
R
InsCnt=1
MaxHopCnt=16
TotalHopCnt=3
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
Reserved
queue occupancy of hop3 (sw1)
queue occupancy of hop2 (sw2)
queue occupancy of hop1 (sw3)

P4 program specification for INT Transit

P4 program is an unambiguous description of a protocol. Here we present a P4 program for an INT Transit device, describing the INT headers, the parsing operations for the headers, and the match-action tables implementing the INT Transit behaviors. The program is written in P4_16 on v1model architecture, and it assumes INT over TCP as the encapsulation protocol.

This program is work in progress.

```
/*************************************************************************
 * header.p4
 *************************************************************************/

/* INT shim header for TCP/UDP */
header intl4_shim_t {
    bit<8>  int_type;
    bit<8>  rsvd1;
    bit<8>  len;
    bit<8>  rsvd2;
}

/* INT tail header for TCP/UDP */
header intl4_tail_t {
    bit<8>  next_proto;
    bit<16> dest_port;
    bit<8>  dscp;
}

/* INT headers */
header int_header_t {
    bit<2>  ver;
    bit<2>  rep;
    bit<1>  c;
    bit<1>  e;
    bit<5>  rsvd1;
    bit<5>  ins_cnt;
    bit<8>  max_hop_cnt;
    bit<8>  total_hop_cnt;
    bit<4>  instruction_mask_0003;  /* split the bits for lookup */
    bit<4>  instruction_mask_0407;
    bit<4>  instruction_mask_0811;
    bit<4>  instruction_mask_1215;
    bit<16> rsvd2;
}

/* INT meta-value headers - different header for each value type */
header int_switch_id_t {
    bit<32> switch_id;
}
```

```
header int_port_ids_t {
    bit<16> ingress_port_id;
    bit<16> egress_port_id;
}

header int_hop_latency_t {
    bit<32> hop_latency;
}

header int_q_occupancy_t {
    bit<8>  q_id;
    bit<24> q_occupancy;
}

header int_ingress_tstamp_t {
    bit<32> ingress_tstamp;
}

header int_egress_tstamp_t {
    bit<32> egress_tstamp;
}

header int_q_congestion_t {
    bit<8>  q_id;
    bit<24> q_congestion;
}

header int_egress_port_tx_util_t {
    bit<32> egress_port_tx_util;
}

/* standard ethernet/ip/tcp headers */
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

/* define diffserv field as DSCP(6b) + ECN(2b) */
header ipv4_t {
    bit<4>  version;
    bit<4>  ihl;
    bit<6>  dscp;
    bit<2>  ecn;

    bit<16> totalLen;
    bit<16> identification;
    bit<3>  flags;
    bit<13> fragOffset;
    bit<8>  ttl;
    bit<8>  protocol;
    bit<16> hdrChecksum;
```

```
    bit<32> srcAddr;
    bit<32> dstAddr;
}

header tcp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<8>  flags;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

struct headers {
    ethernet_t              ethernet;
    ipv4_t                  ipv4;
    tcp_t                   tcp;
    intl4_shim_t            intl4_shim;
    int_header_t            int_header;
    int_switch_id_t         int_switch_id;
    int_port_ids_t          int_port_ids;
    int_hop_latency_t       int_hop_latency;
    int_q_occupancy_t       int_q_occupancy;
    int_ingress_tstamp_t    int_ingress_tstamp;
    int_egress_tstamp_t     int_egress_tstamp;
    int_q_congestion_t      int_q_congestion;
    int_egress_port_tx_util_t int_egress_port_tx_util;
}

/* switch internal variables for INT logic implementation */
struct int_metadata_t {
    bit<16> insert_byte_cnt;
    bit<8>  int_hdr_word_len;
    bit<32> switch_id;
}

struct metadata {
    int_metadata_t   int_metadata;
}


/*************************************************************************
 * parser.p4
 *************************************************************************/

/* indicate INT at LSB of DSCP */
const bit<6> DSCP_INT = 0x1;
```

```
parser ParserImpl(packet_in packet, out headers hdr, inout metadata meta, inout
standard_metadata_t standard_metadata) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            16w0x800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            8w0x6: parse_tcp;
            default: accept;
        }
    }
    state parse_tcp {
        packet.extract(hdr.tcp);
        transition select(hdr.ipv4.dscp) {
            DSCP_INT &&& DSCP_INT: parse_intl4_shim;
            default: accept;
        }
    }
    state parse_intl4_shim {
        packet.extract(hdr.intl4_shim);
        transition parse_int_header;
    }
    state parse_int_header {
        packet.extract(hdr.int_header);
        transition accept;
    }
}

control DeparserImpl(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
        packet.emit(hdr.intl4_shim);
        packet.emit(hdr.int_header);
        packet.emit(hdr.int_switch_id);
        packet.emit(hdr.int_port_ids);
        packet.emit(hdr.int_hop_latency);
        packet.emit(hdr.int_q_occupancy);
        packet.emit(hdr.int_ingress_tstamp);
        packet.emit(hdr.int_egress_tstamp);
        packet.emit(hdr.int_q_congestion);
        packet.emit(hdr.int_egress_port_tx_util);
    }
```

```
}

control VerifyChecksumImpl(in headers hdr, inout metadata meta) {
    Checksum16() ipv4_checksum;
    apply {
        if (hdr.ipv4.hdrChecksum == ipv4_checksum.get({ hdr.ipv4.version,
hdr.ipv4.ihl, hdr.ipv4.dscp, hdr.ipv4.ecn, hdr.ipv4.totalLen,
hdr.ipv4.identification, hdr.ipv4.flags, hdr.ipv4.fragOffset, hdr.ipv4.ttl,
hdr.ipv4.protocol, hdr.ipv4.srcAddr, hdr.ipv4.dstAddr }))
            mark_to_drop();
    }
}

control ComputeChecksumImpl(inout headers hdr, inout metadata meta) {
    Checksum16() ipv4_checksum;
    apply {
        hdr.ipv4.hdrChecksum = ipv4_checksum.get({ hdr.ipv4.version, hdr.ipv4.ihl,
hdr.ipv4.dscp, hdr.ipv4.ecn, hdr.ipv4.totalLen, hdr.ipv4.identification,
hdr.ipv4.flags, hdr.ipv4.fragOffset, hdr.ipv4.ttl, hdr.ipv4.protocol,
hdr.ipv4.srcAddr, hdr.ipv4.dstAddr });
    }
}


/*************************************************************************
 * int_transit.p4: tables, actions, and control flow
*************************************************************************/

#include <core.p4>
#include <v1model.p4>

#include "header.p4"
#include "parser.p4"

control Int_metadata_insert(inout headers hdr,
        in int_metadata_t int_metadata,
        inout standard_metadata_t standard_metadata)
{
    /* this reference implementation covers only INT instructions 0-3 */
    action int_set_header_0() {
        hdr.int_switch_id.setValid();
        hdr.int_switch_id.switch_id = int_metadata.switch_id;
    }
    action int_set_header_1() {
        hdr.int_port_ids.setValid();
        hdr.int_port_ids.ingress_port_id =
            (bit<16>) standard_metadata.ingress_port;
        hdr.int_port_ids.egress_port_id =
            (bit<16>) standard_metadata.egress_port;
    }
    action int_set_header_2() {
        hdr.int_hop_latency.setValid();
```

```
    hdr.int_hop_latency.hop_latency =
        (bit<32>) standard_metadata.deq_timedelta;
}
action int_set_header_3() {
    hdr.int_q_occupancy.setValid();
    hdr.int_q_occupancy.q_id =
        (bit<8>) standard_metadata.egress_qid;
    hdr.int_q_occupancy.q_occupancy =
        (bit<24>) standard_metadata.deq_qdepth;
}

/* action functions for bits 0-3 combinations, 0 is msb, 3 is lsb */
/* Each bit set indicates that corresponding INT header should be added */
action int_set_header_0003_i0() {
}
action int_set_header_0003_i1() {
    int_set_header_3();
}
action int_set_header_0003_i2() {
    int_set_header_2();
}
action int_set_header_0003_i3() {
    int_set_header_3();
    int_set_header_2();
}
action int_set_header_0003_i4() {
    int_set_header_1();
}
action int_set_header_0003_i5() {
    int_set_header_3();
    int_set_header_1();
}
action int_set_header_0003_i6() {
    int_set_header_2();
    int_set_header_1();
}
action int_set_header_0003_i7() {
    int_set_header_3();
    int_set_header_2();
    int_set_header_1();
}
action int_set_header_0003_i8() {
    int_set_header_0();
}
action int_set_header_0003_i9() {
    int_set_header_3();
    int_set_header_0();
}
action int_set_header_0003_i10() {
    int_set_header_2();
    int_set_header_0();
}
```

```
action int_set_header_0003_i11() {
    int_set_header_3();
    int_set_header_2();
    int_set_header_0();
}
action int_set_header_0003_i12() {
    int_set_header_1();
    int_set_header_0();
}
action int_set_header_0003_i13() {
    int_set_header_3();
    int_set_header_1();
    int_set_header_0();
}
action int_set_header_0003_i14() {
    int_set_header_2();
    int_set_header_1();
    int_set_header_0();
}
action int_set_header_0003_i15() {
    int_set_header_3();
    int_set_header_2();
    int_set_header_1();
    int_set_header_0();
}

/* Table to process instruction bits 0-3 */
table int_inst_0003 {
    key = {
        hdr.int_header.instruction_mask_0003 : exact;
    }
    actions = {
        int_set_header_0003_i0;
        int_set_header_0003_i1;
        int_set_header_0003_i2;
        int_set_header_0003_i3;
        int_set_header_0003_i4;
        int_set_header_0003_i5;
        int_set_header_0003_i6;
        int_set_header_0003_i7;
        int_set_header_0003_i8;
        int_set_header_0003_i9;
        int_set_header_0003_i10;
        int_set_header_0003_i11;
        int_set_header_0003_i12;
        int_set_header_0003_i13;
        int_set_header_0003_i14;
        int_set_header_0003_i15;
    }
    default_action = int_set_header_0003_i0();
    size = 16;
}
```

```
    /* Similar tables can be defined for instruction bits 4-7 and bits 8-11 */
    /* e.g., int_inst_0407, int_inst_0811 */

    apply{
        int_inst_0003.apply();
        // int_inst_0407.apply();
        // int_inst_0811.apply();
    }
}

control Int_outer_encap(inout headers hdr,
        in int_metadata_t int_metadata)
{
    action int_update_ipv4() {
        hdr.ipv4.totalLen = hdr.ipv4.totalLen + int_metadata.insert_byte_cnt;
    }
    action int_update_shim() {
        hdr.intl4_shim.len = hdr.intl4_shim.len + int_metadata.int_hdr_word_len;
    }

    apply{
        if (hdr.ipv4.isValid()) {
            int_update_ipv4();
        }
        /* Add: UDP length update if you support UDP */

        if (hdr.intl4_shim.isValid()) {
            int_update_shim();
        }
    }
}


control Int_egress(inout headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata)
{
    action int_transit(bit<32> switch_id) {
        meta.int_metadata.switch_id = switch_id;
        meta.int_metadata.insert_byte_cnt = (bit<16>) hdr.int_header.ins_cnt << 2;
        meta.int_metadata.int_hdr_word_len = (bit<8>) hdr.int_header.ins_cnt;
    }
    table int_prep {
        key = {}
        actions = {int_transit;}
    }

    Int_metadata_insert() int_metadata_insert;
    Int_outer_encap() int_outer_encap;

    action int_hop_cnt_increment() {
        hdr.int_header.total_hop_cnt =
```

```
            hdr.int_header.total_hop_cnt + 1;
    }
    action int_hop_cnt_exceeded() {
        hdr.int_header.e = 1;
    }

    apply{
        if(hdr.int_header.isValid()) {
            if(hdr.int_header.max_hop_cnt != hdr.int_header.total_hop_cnt
                    && hdr.int_header.e == 0)
            {
                int_hop_cnt_increment();
                int_prep.apply();
                int_metadata_insert.apply(
                    hdr, meta.int_metadata, standard_metadata);
                int_outer_encap.apply(hdr, meta.int_metadata);
            } else {
                int_hop_cnt_exceeded();
            }
        }
    }
}

control IngressImpl(inout headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata)
{
    apply{
        /* fill in */
    }
}

control EgressImpl(inout headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata)
{
    Int_egress() int_egress;
    apply{
        // snip
        int_egress.apply(hdr, meta, standard_metadata);
        // snip
    }
}

V1Switch(ParserImpl(),
         VerifyChecksumImpl(),
         IngressImpl(),
         EgressImpl(),
         ComputeChecksumImpl(),
         DeparserImpl()) main;

/* End of Code snippet */
```

In-band Network Telemetry (INT) -- Appendix

October 2017

Appendix 1: An extensive (but not exhaustive) set of Metadata

Switch-level Information

- Switch id
  - The unique ID of a switch (generally administratively assigned). SwitchIDs must be unique within a management domain.
- Control plane state version number
  - Whenever a control-plane state changes (e.g., IP FIB update), the switch control plane can also update this version number in the data plane. INT packets may use these version numbers to determine which control-plane state was active at the time packets were forwarded.

Ingress Information

- Ingress port id
  - The logical port on which the INT packet was received. The semantics (meaning) of this value may differ for individual devices. A device may expose physical and logical port identifiers separately.
- Ingress timestamp
  - The device local time when the INT packet was received on the ***ingress*** physical or logical port.
- Ingress port RX pkt count
  - Total # of packets received so far (since device initialization or counter reset) on the ingress physical or logical port where the INT packet was received.
- Ingress port RX byte count
  - Total # of bytes received so far on the ingress physical or logical port where the INT packet was received.
- Ingress port RX drop count
  - Total # of packet drops occurred so far on the ingress physical or logical port where the INT packet was received.
- Ingress port RX utilization
  - Current utilization of the ingress physical or logical port where the INT packet was received. The exact mechanism (bin bucketing, moving average, etc.) is device specific and while the latter is clearly superior to the former, the INT framework leaves those decisions to device vendors.

Egress Information

- Egress physical port id
  - The ID of the port via which the INT packet was sent out.

- Egress timestamp
    - Device local time capturing when the INT packet leaves the egress port.
- Egress port TX pkt count
    - Total # of packets forwarded so far (since device initialization or counter reset) through the egress physical or logical port where the INT packet was also forwarded.
- Egress port TX byte count
    - Total # of bytes forwarded so far through the egress physical or logical port where the INT packet was forwarded.
- Egress port TX drop count
    - Total # of packet drops occurred so far on the egress physical or logical port where the INT packet was forwarded.
- Egress port TX utilization
    - Current utilization of the egress port via which the INT packet was sent out.

Buffer Information

- Queue id
    - The id of the queue the device used to serve the INT packet.
- Instantaneous queue length
    - The instantaneous length (in bytes, cells, or packets) of the queue the INT packet has observed in the device while being forwarded. The units used need not be consistent across an INT domain, but care must be taken to ensure that there is a known, consistent mapping of {device, queue} values to their respective unit {packets, bytes, cells}.
- Average queue length
    - The average length (in bytes, cells, or packets) of the queue via which the INT packet was served. The calculation mechanism of this value is device specific.
- Congestion status
    - The ratio of the current queue length to the configured maximum queue limit. This value is used primarily to determine how much space is left in the queue.
- Queue drop count
    - Total # of packets dropped from the queue

Changelog / Release History

Date	Changes / Notes
2015-09-28	initial release
2016-06-19	Changes: ● Updated section 5.3.2, the Length field definition of VXLAN GPE shim header, to be consistent with the example of page 15.
2017-10-17	Changes: ● Introduced INT over TCP/UDP (section 5.3.1 and new example) ● Removed BOS (Bottom-Of-Stack) bit at each 4B metadata, from the header definition and examples ● Updated the INT instruction bitmap and the meaning of a few instructions (section 5.3.4) ● Moved the INT transit P4 program from Appendix 2 to the main section 6. Re-wrote the program in p4_16.